

# Comments on the Reliability of Lawson and Hanson's Linear Distance Programming Algorithm: Subroutine LDP \*

Alan Rufty

Naval Surface Warfare Center Dahlgren  
Attn: Alan Rufty  
5443 Bronson Road, Suite 112  
Dahlgren, VA. 22448-5159

February 1, 2008

## Abstract

This brief paper: (1) Discusses strategies to generate random test cases that can be used to extensively test any Linear Distance Program (LDP) software. (2) Gives three numerical examples of input cases generated by this strategy that cause problems in the Lawson and Hanson LDP module. (3) Proposes, as a standard matter of acceptable implementation procedures, that (unless it is done internally in the software itself, but, in general, this seems to be much rarer than one would expect) all users should test the returned output from any LDP module for self-consistency since it incurs only a small amount of added computational overhead and it is not hard to do.

**ACM Categories and Subject Descriptors:** G.4 [**Mathematical Software**]: Reliability and robustness; D.2.5 [**Testing and Debugging**]: Testing tools (e.g., data generators, coverage testing)

**Key words:** Linear Distance Programming, Matrix Inequality, Kuhn-Tucker Theorem

**AMS subject classification (2000):** Primary 15A39. Secondary 15A45

## 1 Introduction

Given some  $m \times n$  matrix  $\mathbf{G}$  and  $m$ -vector  $\mathbf{h}$ , the Linear Distance Programming (LDP) problem consists of finding the  $n$ -vector  $\mathbf{X}$  of minimum norm that satisfies the following system of inequality constraints (when a solution exists):

$$\mathbf{GX} \geq \mathbf{h} . \quad (1)$$

Here the phrase “ $\mathbf{X}$  is of minimum norm” means that  $|\mathbf{X}|$  is to be minimized, subject to the set of constraints given by (1) and provided that these constraints are consistent.

A properly implemented LDP algorithm must thus:

1. find the  $\mathbf{X}$  of minimum norm, when the system of constraints given by (1) is consistent or
2. return a flag indicating that the set of constraints given by (1) is inconsistent and thus that no solution exists.

---

\*Approved for public release.

The very well known and extensively utilized Lawson and Hanson software module “Subroutine LDP” is an implementation of this LDP problem [1, 2]. The distributions of the Lawson and Hanson algorithm(s) are generally in Fortran and can be ported from the web site netlib at:

<http://netlib.org/lawson-hanson/all> .

These distributions are also available at numerous other web sites such as

<http://netlib.bell-labs.com/lawson-hanson/index.html> .

When the inequality constraints are determined to be consistent, the Lawson and Hanson module LDP sets the flag MODE to 1 and returns a solution vector  $\mathbf{X}$ . Alternatively, if the constraints are determined to be inconsistent, the Lawson and Hanson module LDP sets the flag MODE to 4 indicating that there is no solution. The prime concern here is the relatively rare occurrence of a MODE = 1 return with an associated solution vector  $\mathbf{X}$  that fails to satisfy the set of constraints (1). Although these problem cases seem to be associated with issues in handling significant digits, the erroneous returned solutions may, in fact, produce numerically sizeable violations of the specified inequality constraints.

These types of problem examples were uncovered using a strategy of systematically generating copious numbers of random cases. This general strategy is outlined in Section 2 and three specific numerical examples generated by this strategy that cause problems and have small values of  $m$  and  $n$  are given in Section 3 and then discussed in Section 4.

Although the Lawson and Hanson software LDP module [1, 2] is of primary interest here, the general checkout strategies and recommendations apply to other LDP implementations as well. In particular similar testing was performed on the NSWC Mathematics Library Subroutine LSEI [3], which can be used to handle LDP problems. This testing revealed that LSEI was also prone to yield a MODE = 1 return under rare circumstances when the set of constraints (1) was inconsistent; however, the properties of the  $\mathbf{G}$  matrices that caused problems appeared to be different for LSEI and LDP. In this regard, the remark at the bottom of page 393 of [3] may be relevant and is worth repeating here:

1. “LSEI may perform poorly if the norms of the rows of  $\mathbf{A}$  and  $\mathbf{E}$  differ by many orders of magnitude, or if the norms of the rows of  $\mathbf{E}$  are exceedingly small.”

Given that problems were found for the only two LDP algorithms tested and because it is clearly easy to do and entails only a small added computational burden, as a matter of course, it is recommended that users of any LDP software module should automatically check the constraint condition (1) of all returned solutions when the software module indicates that there is one.

## 2 Random Case Generation Strategy

One significant point is that random cases that are guaranteed to satisfy some inequality constraint or other can be generated systematically. First, consider the set of inequality constraints that results for the choice  $\mathbf{h} = 0$ :

$$\mathbf{GX} \geq 0 . \quad (2)$$

Clearly, in this case,  $\mathbf{X} = 0$  is always the solution to the LDP problem since  $|\mathbf{X}| = 0$ .

It is easy to generalize this solution. Towards that end consider the set of conditions

$$\mathbf{G}(\mathbf{X} - \mathbf{X}_0) \geq 0 , \quad (3)$$

where  $\mathbf{X}_0$  is some (randomly) chosen vector. Clearly (3) always specifies a set of consistent constraints since  $\mathbf{X} = \mathbf{X}_0$  is always a valid solution to the constraint conditions. Some other choice of  $\mathbf{X}$  with  $|\mathbf{X}| < |\mathbf{X}_0|$  may exist so  $\mathbf{X} = \mathbf{X}_0$  may not be the actual solution to the LDP problem, but at least a solution is guaranteed.

Thus, for any randomly chosen values of  $m$  and  $n$ , if  $\mathbf{G}$  is a randomly generated  $m \times n$  matrix and  $\mathbf{X}_0$  is a randomly generated  $n$ -vector, then provided the  $n$ -component  $\mathbf{h}$  vector is computed by

$$\mathbf{h} := \mathbf{G}\mathbf{X}_0 , \quad (4)$$

a consistent solution to (1) always exists.

This whole procedure can be carried one step further to obtain a more general set of self-consistent equations of the form (1). Let  $\mathbf{A}$  be a general (i.e., randomly generated)  $l \times m$  matrix and  $\mathbf{B}$  be an arbitrary invertible (i.e., also possibly randomly generated)  $n \times n$  matrix. Then (3) yields the following form

$$\mathbf{A}\mathbf{G}\mathbf{B}\mathbf{B}^{-1}(\mathbf{X} - \mathbf{X}_0) \geq 0 , \quad (5)$$

which can be rewritten as

$$\mathbf{G}'\mathbf{X}' \geq \mathbf{h}' \quad (6)$$

where

$$\mathbf{G}' := \mathbf{A}\mathbf{G}\mathbf{B} \quad \text{and} \quad \mathbf{h}' := \mathbf{A}\mathbf{G}\mathbf{X}_0 . \quad (7)$$

Here the solution  $\mathbf{X}'$  is related to  $\mathbf{X}$  by  $\mathbf{X}' = \mathbf{B}^{-1}\mathbf{X}$  and the inclusion of  $\mathbf{B}$  here allows for additional freedom in the choice of  $\mathbf{X}$  since it helps to de-link the solution vector from  $\mathbf{X}_0$ .

By generating a random  $l, m$ , and  $n$  and then appropriate random matrices  $\mathbf{A}$ ,  $\mathbf{B}$  and  $\mathbf{G}$ , along with a random vector  $\mathbf{X}_0$ , it is easy to test out any LDP algorithm for the case of self-consistent constraints using (7). Notice that in practice it is easy to generate a non-singular  $\mathbf{B}$  matrix with random elements: Simply generate completely random  $\mathbf{B}$  matrices and then test the result to see if  $|\mathbf{B}| = 0$  and then simply regenerate the  $\mathbf{B}$  matrix if it is. (Appropriate scaling factors, as well as other minor implementation issues are obvious from the examples in Section 2.)

Observe that since the existence of a solution in the above derivation was predicated on the existence of equalities [i.e., the  $\geq$  and not  $=$  in, say (1) or (7)], the region where the constraints are consistent may consist of only a single point and thus (due to round-off or other factors) the LDP algorithms may potentially experience some problems in finding and testing for this one point. One way to enlarge this interior feasibility region is to replace (1) by

$$\mathbf{G}\mathbf{X} \geq \tilde{\mathbf{h}} , \quad (8)$$

where  $\tilde{\mathbf{h}} = \mathbf{h} - \mathbf{C}$ , for some constant or random vector  $\mathbf{C}$ , all of whose components are positive. It is also easy to modify the above random case generation strategy so that inconsistent constraints are produced. One strategy, for example, is to simply replace  $\tilde{\mathbf{h}}$  in (8) by  $\tilde{\mathbf{h}} = \mathbf{h} + \mathbf{D}$  for some random vector  $\mathbf{D}$  with sufficiently large components.

### 3 Numerical Examples of Randomly Generated Problem Cases

Three numerical examples of inconsistent inequality constraints are given below. When the input for each of these cases is used in the Lawson and Hanson Fortran Subroutine LDP implemented on a SUN workstation using *double precision*, a  $\text{MODE} = 1$  return results and a candidate solution vector is returned, whereas there should be a  $\text{MODE} = 4$  return indicating that no solution exists. Specific problem cases are by nature somewhat finicky and difficult to replicate and may not cause similar problems on other computer systems—they are, in fact, highly dependent on the number of internal significant digits used for the inputs, as well as in the internal computations themselves. (SUN architecture reuses the mantissa when going from single to *double precision* so there are generally 16 or so significant digits of internal representation rather than the 14 or so one might normally expect.) This, however, does not mean that the problem is tied to only

SUN systems or that they are more common on SUN systems; rather, it means that different specific cases will cause problems on different systems. Given this state of affairs, the reader may well not be able to use the input cases below to generate inconsistent returns from Subroutine LDP and may thus have to use the strategies outlined above to generate such cases randomly. [Generally,  $\mathbf{A}$  and  $\mathbf{B}$  can be taken to be identity matrices, but a certain number of columns of the random  $\mathbf{G}$  should be zero and since (for what might be typical reasonable parameter settings designed to generate such cases and an associated well-chosen random test case generation methodology) the incidences of such problem cases might usually be only approximately one in a thousand or so, probably at least 50,000 or so cases of various types and with parameter settings should normally be generated.] In passing, it is worth noting that the problems addressed here are not tied specifically to Fortran: a C programming language version of the Lawson and Hanson LDP algorithm was also tested and displayed the same problems.

Finally, before considering the examples, two observations are in order. First, notice that the Lawson and Hanson in-line function “DIFF” is used in their LDP implementation to help prevent internal significant digit masking, but this, in itself, does not appear to resolve the problems indicated here. Second, when these same inputs are used with a *type quad* internal Fortran version (which corresponds to approximately 32 internal digits of representation), then they all yield  $\text{MODE} = 4$  returns, but it seems clear that simply going to a *type quad* implementation will not solve the problem (although it will lessen the incidence of it)—it will only make counter-examples harder to find.

Although the cases  $m \gg n \gg 0$  are the ones that generate the most problem cases, for simplicity only three cases with  $m = 4$  and  $n = 2$  are considered.

Case 1

$$\mathbf{G} = \begin{bmatrix} -89.20509815216064 & 0.0000000000000000 \\ 74.79768991470337 & 0.0000000000000000 \\ 66.23740792274475 & 0.0000000000000000 \\ -18.51919293403625 & 0.0000000000000000 \end{bmatrix} \quad \mathbf{h} = \begin{bmatrix} -12073.43407295207 \\ 10123.19482867013 \\ 8350.549301112449 \\ -24612.94532321187 \end{bmatrix} \quad (9)$$

Case 2

$$\mathbf{G} = \begin{bmatrix} 81.82253837585449 & 0.0000000000000000 \\ -74.02672171592712 & 0.0000000000000000 \\ 0.0000000000000000 & -17.36225485801697 \\ -89.47155475616455 & 0.0000000000000000 \end{bmatrix} \quad \mathbf{h} = \begin{bmatrix} -77004.09890544150 \\ 69248.37468031116 \\ 11241.52852765946 \\ 84233.37742495652 \end{bmatrix} \quad (10)$$

Case 3

$$\mathbf{G} = \begin{bmatrix} -3.057897090911865 & 0.0000000000000000 \\ 4.310655593872070 & 0.0000000000000000 \\ 39.13614749908447 & 0.0000000000000000 \\ 84.55699086189270 & 0.0000000000000000 \end{bmatrix} \quad \mathbf{h} = \begin{bmatrix} 2192.778913749731 \\ -3422.354440768027 \\ -28760.98260603488 \\ -60562.89687439907 \end{bmatrix} \quad (11)$$

## 4 Discussion of Randomly Generated Problem Case Outputs

For each of the above examples, it is easy to verify directly that the system of constraints implied by (1) is consistent; however, when the input for each of these examples was used by the author in the Lawson and Hanson Subroutine LDP implemented in *double precision* on a SUN system, a  $\text{MODE} = 1$  return resulted with an improper solution vector. Specifically the returned solution vectors were:

Case 1

$$\mathbf{X} = \begin{bmatrix} -.3750000000000000 \\ .0000000000000000 \end{bmatrix} \quad (12)$$

Case 2

$$\mathbf{X} = \begin{bmatrix} .6074218750000000 \\ .0000000000000000 \end{bmatrix} \quad (13)$$

Case 2

$$\mathbf{X} = \begin{bmatrix} .4570312500000000 \\ .0000000000000000 \end{bmatrix} \quad (14)$$

For these cases, direct numerical testing shows that the problems seem to be in the Subroutine LDP itself and not in the other subroutines it calls. More specifically, using Lawson and Hanson's notation [1, 2], except for the use of bold face type for vectors and matrices, from the returns supplied to Subroutine LDP from Subroutine NNLS (which LDP calls), the basic vectors and matrices  $\mathbf{E}$ ,  $\mathbf{r}$ ,  $\mathbf{p}$  and  $\hat{\mathbf{x}}$  can be determined. {Specifically, see equations (23.28) through (23.34) in [1] or [2].} Using these computed quantities it easy to show that due to round-off and/or masking the basic assumptions in proving the validity of the LDP algorithm do not hold; For example, that some of the components of the  $\mathbf{p}$  vector are negative. For these specific inputs all of these inconsistencies go away when *type quad* variables are used, but as indicated above, one would expect to be able to find similar cases when a full set of *type quad* random cases (one tricky part here appears to be finding a full-up *type quad* random number generator). Finally, with regards to *type quad* Fortran implementations, it is perhaps worth noting that there is a SUN *f77* compiler option that automatically converts from *type double* to *type quad*.

## References

- [1] C. Lawson and R. Hanson, *Solving Least Squares Problems*, First Edition, Prentice–Hall, Englewood Cliffs, N. J., 1974.
- [2] C. Lawson and R. Hanson, *Solving Least Squares Problems*, Classics in Applied Mathematics **15**, SIAM Publications, Philadelphia, P.A., 1995.
- [3] Alfred H. Morris, Jr., *NSWC Library of Mathematics Subroutines*, Classics in Applied Mathematics **NSWC TR-92/425**, Naval Surface Warfare Center Dahlgren Division, Dahlgren, Virginia 22448-5000.